CP Debugging Needs

Task TWP1.1: Clarification of functionalities. Selection of the tools

Deliverable: report D.WP1.1.M1.1- Part1

Release: n.3

April 30, 1997

Massimo Fabris
ICON s.r.l.
Viale dell'Industria 21 - Verona
Italy
email: fabris@icon.it

Contents

1. INTRODUCTION 3

2. A QUESTIONNAIRE ON DEBUGGING NEEDS 4

2.1 Individual information: Q1 5

2.2 Constraint programming experience: Q2 7

2.3 Type of applications developed: Q3 10

2.4 Bug symptoms: Q4 14

2.5 Debugging tools used: Q5 15

2.6 Type of bugs and their frequency: Q6 18

2.7 Difficulties encountered: Q7 20

2.8 Your ideal C(L)P debugging environment: Q8 22

2.9 Comments and examples: Q9 26

3. DEBUGGING NEEDS 27

3.1 Correctness debugging 28

3.2 Performance debugging 28

4. ADDRESSING THE NEEDS 29

4.1 Graphical tools 29

4.2 Assertion based tools 30

4.3 Declarative diagnosis tools 30

5. CONCLUSIONS 31

6. REFERENCES 32

1. Introduction

Conventional programming techniques are not well suited for solving many highly combinatorial industrial problems, like scheduling, decision making, resource allocation or planning. **Constraint Programming** (CP), an emerging software technology, offers an original approach allowing for efficient and flexible solving of complex problems, through combined implementation of various constraint solvers and expert heuristics. Its applications are increasingly elded in various industries.

One of the main features of CP is a new approach to software production: the same program is progressively improved at each step of the development cycle, from the first prototype until the final product. This makes debugging the cornerstone of CP technology. Existing debugging tools reveal to be ineffective in most industrial situations and tools developed for imperative or functional programming are not adapted to the context of CP. The main reasons are that the huge numbers of variables and constraints makes the computation state difficult to understand, and that the non deterministic execution increases drastically the number of computation states which must be analysed. As recognised in [CFGG95] “constraint program are particularly hard to debug, in that it is practically impossible to follow constraint propagation step by step for any non-toy program”.

The main objective of the **DiSCiPl** project is “*to define, to implement and to assess novel and effective debugging systems for Constraint Programming”*. We consider debugging in a broad sense: it concerns both validation aspects (to build a correct application) as well as methodological aspects (to find the best solution to a problem by better understanding of constraint solver behaviour). To satisfy these objectives, tools must locate and explain bugs, and graphic tools must help interpreting program behaviour and results. To reach these goals, we need a clear picture of what is currently used (*be novel*) , what is needed by CP programmers and can be reasonably done on existing widely-used platforms (*be effective*). Novelty and effectiveness were the fundamental objectives of the definition of the DiSCiPl consortium, which includes four academics members and four industrial members. A wider industrial participation is guaranteed by the members of the Advisory Committee.

The goal of the task T.WP1.1 *“Clarification of functionalities. Selection of the tools”* is to identify clearly the needs of the programmers in terms of debugging, during the whole life-cycle of a CP-based software project, and to select the tools which will be developed in the realm of the project to address these needs. This report focuses on the identification of the needs. The instrument to collect these needs was a questionnaire. This questionnaire has been distributed among the members of the consortium and of the advisory committee, and through the most important mailing lists related to Constraint Programming.

The structure of the report is the following: section two fully describes the questionnaire and summarise the answers. Section three discusses how the multiple debugging needs emerging from the questionnaire can be collapsed in two categories of debugging. Section four explains how the DiSCiPl project will tackle these two categories. Section five contains some conclusive remarks.

2. A questionnaire on debugging needs

The best way to have a picture of current debugging needs is asking people who day by day develop CP programs. The simplest way to do it is using a questionnaire. The questionnaire has been circulated in two different times:

- firstly it was circulated only among partners and members of the advisory committee. We collected fifteen complete answers, some of which summarise the experience of many persons working at the same site. The main reason for doing so was time constraint: we needed a reliable picture in a short time, to understand if our first ideas about new tools and techniques adhered to real needs. We were convinced that we could have found a good starting point form these first answers because of the very high experience (theoretical and practical) of these selected persons. It might be argued that it is not fair to ask for a problem definition to the persons who must solve it. However, over the fifteen answers, only two came from persons directly involved in the study and development of new tools;
- in a subsequent period the questionnaire was circulated world-wide through the main mailing lists concerning Constraint Programming. A web version was made available on the site of the project. At the end, we collected 21 answers, giving a global sample of 35 filled questionnaire.

We analysed the data considering two samples:

1. answers collected inside the consortium. These form the **DiSCiPl Sample**;
2. the totality of the answers. These form the **Global Sample**.

In the following we report separately about the two samples whenever they lead to perceptible differences. Another type of aggregation is based on the distinction between academic and industrial institutions: goals, platforms, activities and problems are different in these two environments, hence it is worth to describe these differences.

The questionnaire is composed by nine questions. Question Q1 concerns individual information. Questions Q2 ... Q3 describe the CP experience of the person filling the questionnaire. Questions Q4 ... Q7 describe several aspects of a bug: symptoms, tools used to locate it, frequency of appearance, fixing difficulty. Questions Q8 ... Q9 ask for the main features of an ideal debugger. All questions propose a choice on a space of alternatives. Comments, ideas and examples were encouraged and we collected many ones. Most of them are a significant contribution and we include them in the report (for confidential purposes, we do not quote the authors full names). Thanks to all these persons for their help. Special thanks to Eric Vetillard for the strong help in the definition of the questionnaire, and to Peter Chan and Trijntje Cornellisens for proposing some interesting questions.

2.1 *Individual information: Q1*

This question concerns general individual information about the person answering to the questionnaire:

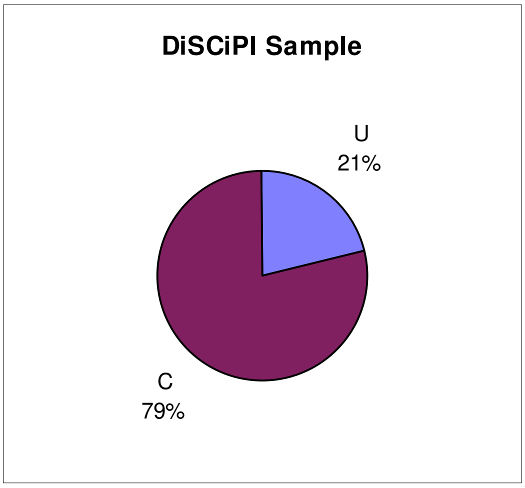
a. Name

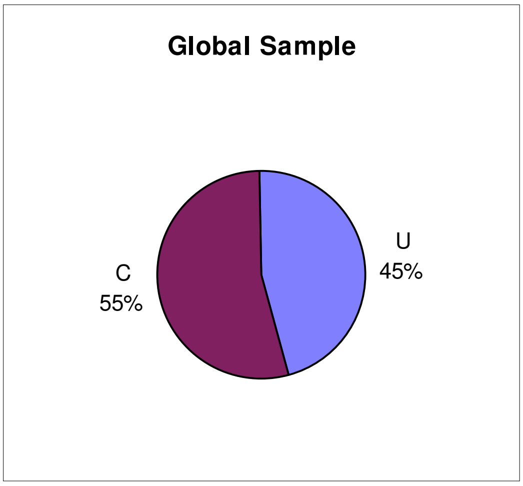
b. Function

c. Institution

d. Type of institution (Company, Academic)

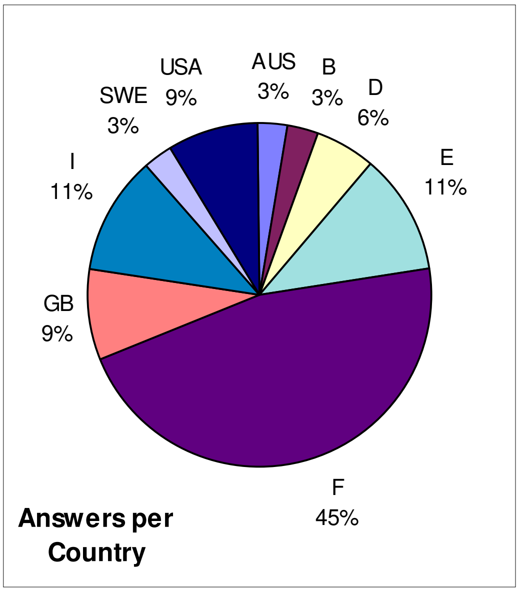
The first aggregation of the samples concerns which type of institution the answering persons belong to. C stands for company, while U stands for University. As we shall see, this distinction is sometimes relevant to give a better interpretation of the data.

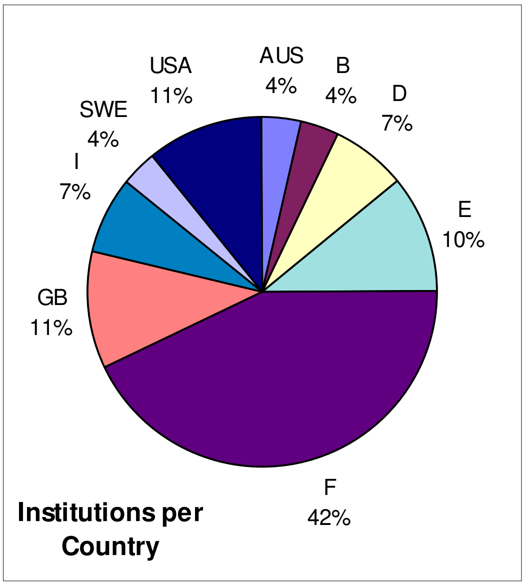




As we can see the DiSCiPl sample is somehow dominated by people coming from industry. This dominance has been re-equilibrated in the Global Sample.

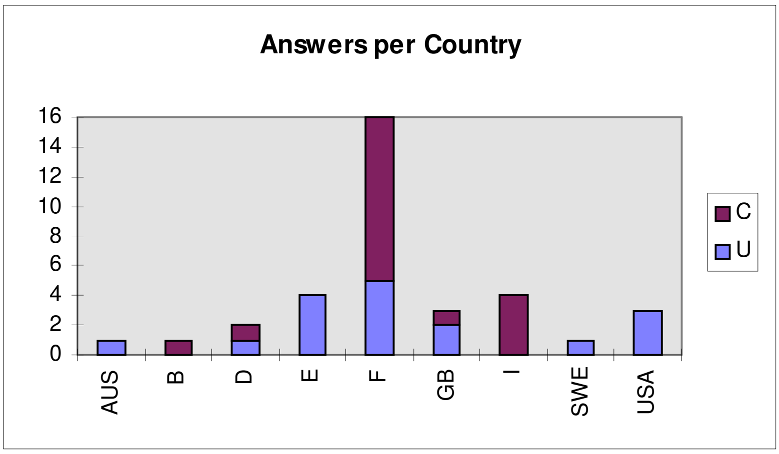
The second extracted information concerns the geographical distribution of the answers. Here we are interested in the location of the institution, not in the nationality of the person. The first figure is about the percentage of answers per country, the second figure is about the percentage of involved institutions per country.

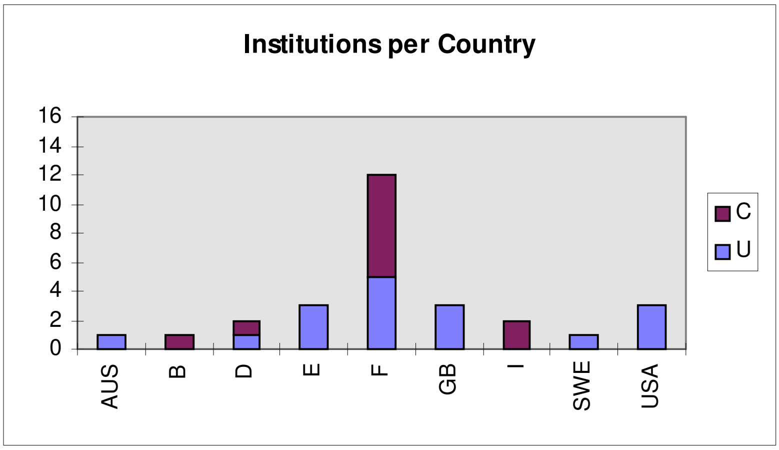




About 85% of the answers come from a european country. It is worth noting the big share of people coming from French universities and companies.

The next figures give the numbers of the *answers per country* and *institutions per country* highlighting the difference between university and industry:





2.2 *Constraint programming experience: Q2*

This section of questions is devoted to describe the constraint programming experience of the interviewed persons. We consider factors as:

- the number of years spent developing CP programs;
- if these programs were applications or prototypes;
- the type of CP platform used to write them.

2.2.1 Programming experience

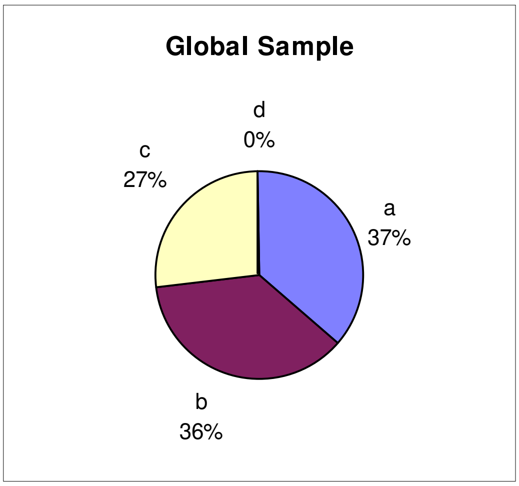
The question about the programming experience provides four mutually exclusive answers:

a. Over 5 years

b. 3 to 5 years

c. 1 to 2 years

d. Less than 1 year



About 70% of the sample has been programming CP languages for more than 3 years. The 37% of them has quite a solid experience (more than 5 years). Nobody declared less than one year. These data confirm the strong competence of the people interviewed.

2.2.2 Development experience

The goal of this question is to verify the type of development experience of the interviewed persons. Many of them gave multiple answers to this question, but they all obeyed the following rule: who has developed at least one operational application, he/she developed several prototypes before. Hence we applied this rule to obtain a set of mutually exclusive answers.

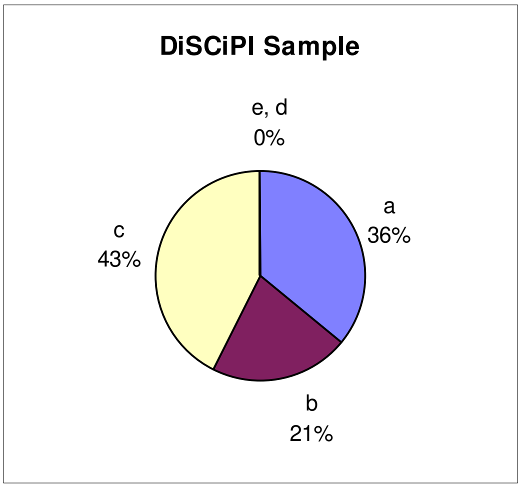
a. Several operational applications

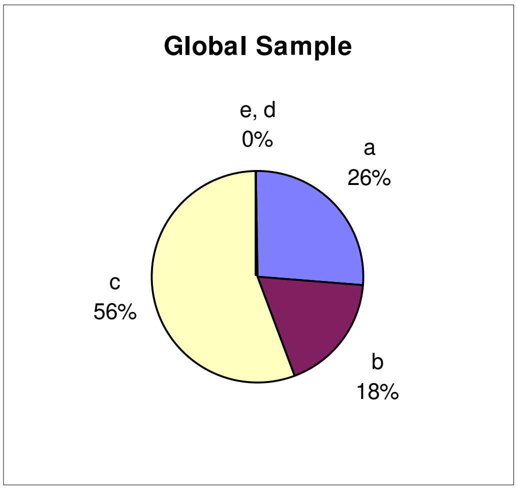
b. One operational application

c. Several prototypes

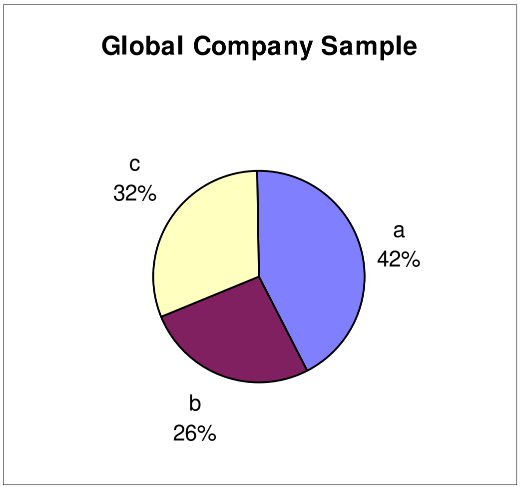
d. One prototype

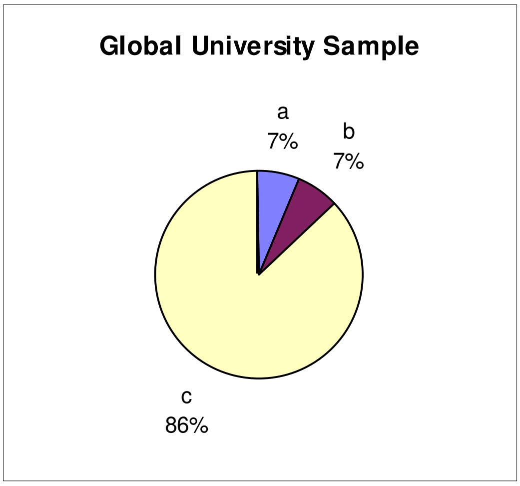
e. No current exploitation





Taking a look at the DiSCiPl sample, we can see that more than 50% of the sample developed at least one operational application, while the 100% developed more than one prototype. On the global sample we have a slight decrease of experience on operational applications. This is due to the stronger contribute of universities in the global sample. If we consider the global sample restricted to universities and companies, we have a better picture of their activities.





It should be stressed that it is not necessarily true that developing only prototypes conveys fewer CP experience. A prototype is very often the kernel solver of the problem, which is the hardest and most important part of the application. Obviously, academic users often develop only prototypes for studying purposes. On the other hand, even if building the rest of the application is much more a standard development matter, this gives a global view on the maturity of the used platform. As it will be clear in the following, developing real-world applications rises relevant issues as data consistency, maintenance of the application and memory consumption.

2.2.3 CP languages used

We wanted to measure the experience of our test sample on different CP languages and platforms. The question was not about the currently used platform: in general each person indicated more than one.

a. BNR-Prolog

b. CHIP

c. CLP(R)

d. Eclipse

e. Ilog Solver

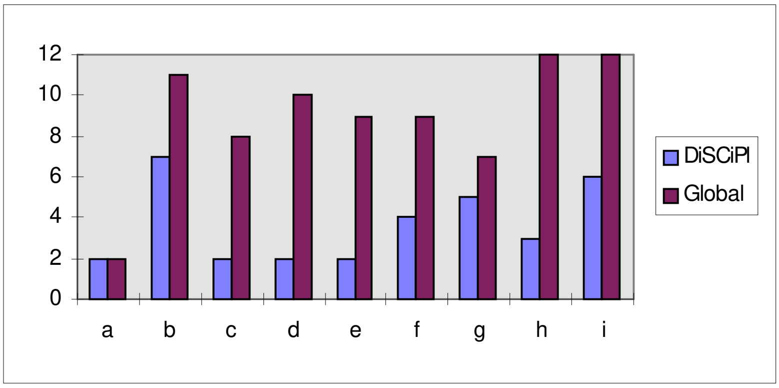
f. Prolog III

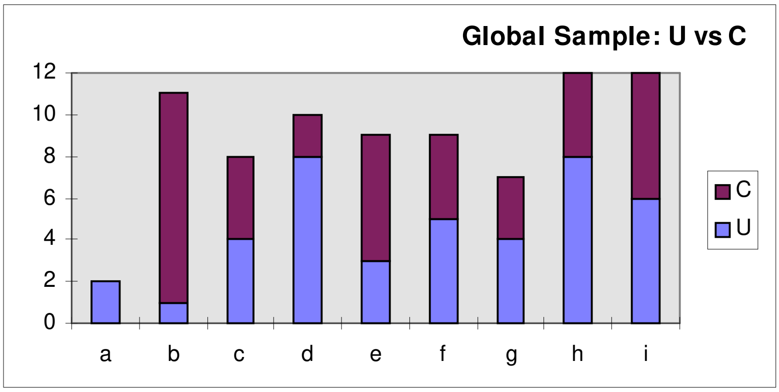
g. Prolog IV

h. CLP(FD)

i. Other

The commercial platforms CHIP and PrologIV, which are the target of the project, and ILOG solver are widely used. Academic platforms like CLP(FD) , CLP(R) and Eclipse are used both for research purpose and for training of students: they are somehow a common background of researchers and software engineers.





The voice "other" includes CHARME (Bull), SICStus, CIAO, Oz, CHR, cc(FD), Claire, Lisp and C++ CP environments.

2.3 *Type of applications developed: Q3*

This section describes in further detail the applications/prototypes developed. The arguments are

- type of problems faced;
- type of solver used;
- size of the resulting problems in terms of lines of code, number of domain variables and constraints, memory usage and so on.

2.3.1 Type of problems

This question provides a snapshot of the application area of CP technology. Planning, scheduling and resource allocation are the main targets, but significant slices are covered by other optimisation problems and more general ones. The data source is the global sample. No significant distinction was evident between the global sample and the DiSCiPl Sample, or between universities and companies.

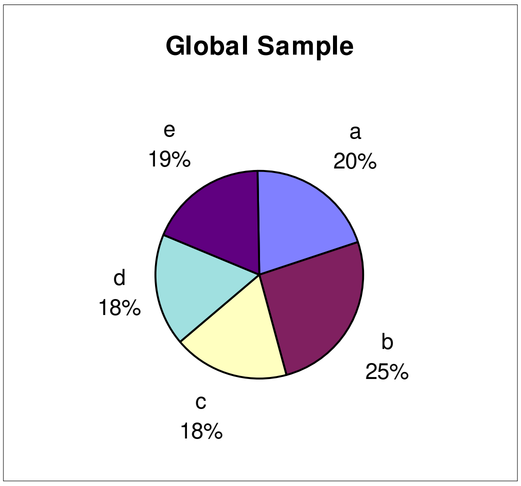
a. Planning

b. Scheduling

c. Resource Allocation

d. Other optimisation { }

e. Other problems



Other reported application areas include: chemical modelisation, 2D placement with complex constraints, 3D placement in a non-convex volume, layout of mechanical objects, time-tabling, network optimisation, operational control, configuration.

2.3.2 Size of problems

This section tries to frame some dimensional parameters of CP applications and prototypes.

Because of the strong difference over the final objective of the developed applications, we separated answers in two categories: those coming from industrial users (C) and those from academic users (U). We report data of both samples, although the global one seem to be more reliable because of a better distribution of answers coming from universities and companies.

We asked to estimate the largest and average value of each parameter. We computed the average of the largest value reported and the average of the average value reported. For completeness, we indicate also the absolute maximal value.

2.3.2.1 Problems Formulation

The first dimensional parameter concerns the constraint model. We asked the largest and average number of (domain) variables and constraints. By (domain) variables we intend variables belonging to the constraint model formulation

a. Largest number of (domain) variables

b. Average number of (domain) variables

c. Largest number of constraints

d. Average number of constraints

DiSCiPl Sample

	a	b	c	d
C	27.000	1.100	184.000	2.600
U	2.550	50	5.000	20
Max	200.000	-	1.000.000	-

Global Sample

	a	b	c	d
C	18.000	1000	114.000	2.000
U	3500	950	100.000	2000
Max	200.000	-	1.000.000	-

This question does not distinguish between “local constraints” and global constraints. A local constraint involves few variables and expresses a simple condition. Propagation inside a local constraint can be traced and understood. A modelisation based on local constraint often uses a lot of them (hundreds of thousand or millions). A global constraint works on sets of variables whose cardinality can be relevant, and expresses complex conditions which must hold among these variables. Propagation inside a global constraint can hardly be traced and understood. Modelisation based on global constraints often uses few of them (at most hundreds), sometimes even only one. They are used as building blocks in the modelisation and can be linked together in multiple ways (see [SK95]).

2.3.2.2 Code size

The third dimensional parameter concerns the code size in number of lines.

a. Size of the largest software developed

b. Size of the average software developed

DiSCiPl Sample

	a	b
C	14.200	3.600
U	300	75
Max	60.000	-

Global Sample

	a	b
C	14.600	3.800
U	4000	300
Max	60.000	-

2.3.2.3 Memory usage

The fourth dimensional parameter concerns the memory usage in Mbytes. Memory consumption of CP applications can be considerable in certain instances of problems where storing all domain variables and constraints involves a relevant dimension of the global stack.

a. Memory usage of the largest software developed

b. Memory usage of the average software developed

DiSCiPl Sample

	a	b
C	67	13
U	19	1
Max	100	-

Global Sample

	a	b
C	60	12
U	57	16
Max	120	-

2.3.3 Type of solvers used

The type of solver used is another fundamental data to be considered in the process of tools selection.

a. Simple finite domains

b. Finite domains with global constraints

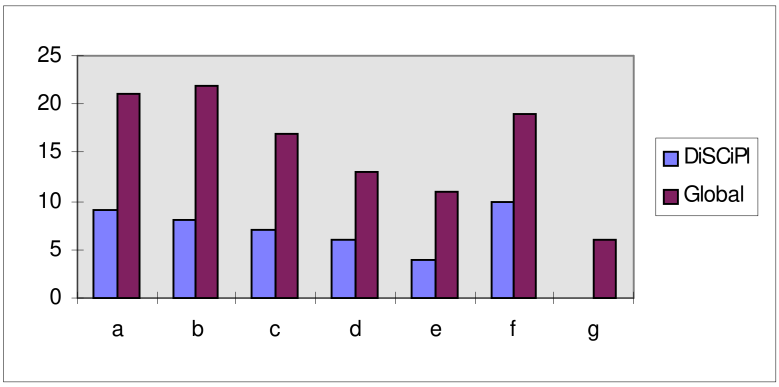
c. Linear

d. Intervals

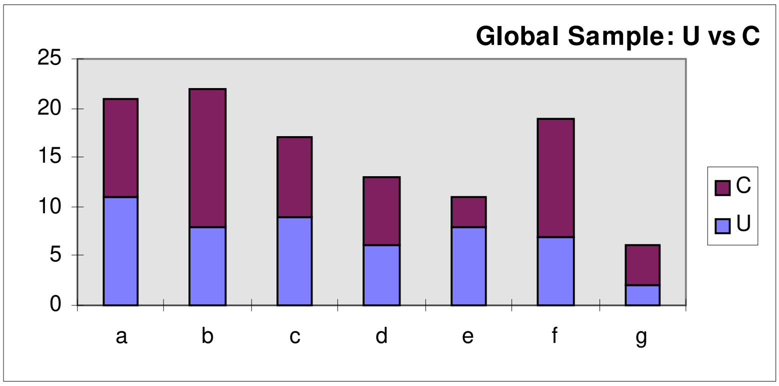
e. Booleans

f. Trees and lists (i.e. Prolog)

g. Other



As expected, finite domains and linear solvers are the most widely used. Prolog and its data structures play a central role in the development of the applications.



2.4 Bug symptoms: Q4

This section enters the hearth of the questionnaire. The focus is on bug symptoms, that is the signals that something in the program does not behave as expected.

The set of proposed symptoms was:

a. Compile/consult time error

b. Runtime error

c. Wrong answers

d. Missing answers or no answers

e. Long to find answers

f. Others

The reader rated the frequency of each symptom according to the following scale:

0: Never

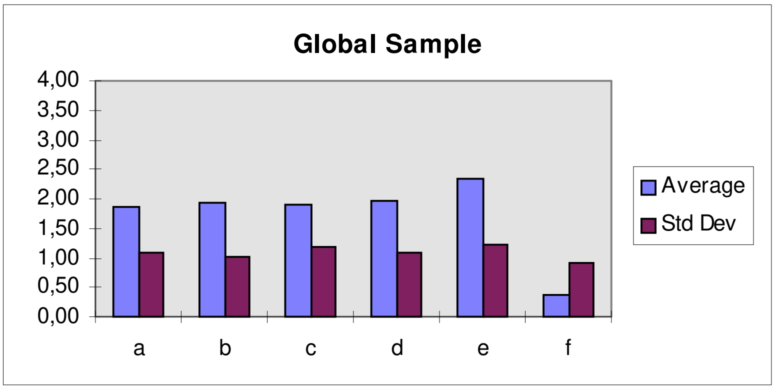
1: Seldom

2: Sometimes

3: Frequently

4:Very frequently

We report the average of the rated frequency and also the standard deviation to measure how far the individual values are from the average.



All the proposed symptoms have been recognised to be equally valid: they have been reported to happen *sometimes* in the average. The stronger difference between U and C was registered on the *e)* symptom leading to an average of 2.17 (std 1.38) of the former vs an average of 2.53 (std 0.96) of the latter.

Note that symptoms a) and b) are probably due to typos in the program text or wrong constraint declarations. Symptoms c) and d) are due to a wrong modelisation, while e) is a symptom of a performance problem.

Comments

A. C. and ***A. F.***: running out of memory is a frequent symptom of bug not mentioned in the questionnaire.

2.5 Debugging tools used: Q5

After the symptom of a bug appears, the programmer is left with the job to find and fix it. Which are the debugging tools that are currently used by CP programmers? We asked the frequency of use of some standards tools or methods to debug a program, or to indicate any alternative.

The proposed tools/methods are:

a. Tracing on standard output (with write)

b. Console tracing (similar to dbx)

c. Source oriented program tracing (similar to dbxtool)

d. Constraint visualisation

e. Static program checking

f. Formal methods

g. Other

The scale is the same as in the previous question:

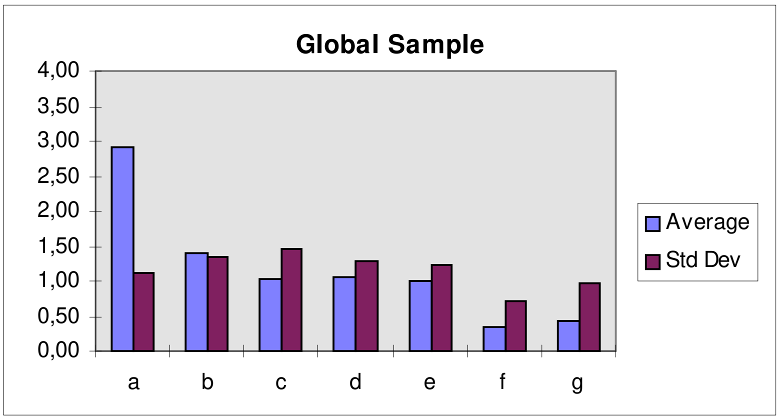
0: Never

1: Seldom

2: Sometimes

3: Frequently

4: Very frequently



Tracing on the output is the most frequently used method. All other methods are not very used, at least in the average. This probably means that the actual CP debugging tools are inadequate. Traditional debuggers (box model or source oriented) should be improved. Constraint visualisation is not very used. This might give the misleading impression that this method is not useful. However, as we shall see, there is a great demand for "visual-graphical debuggers". Constraint visualisation is not very used for this reason: inspite of all facilities, constraint visualisation requires a certain amount of experience and programming work that not all programmers have or can pay. There is hence a demand for graphical debugging tools which are professionally developed and easy to use.

Comments

M. C.: the CHIP environment provides a console debugger allowing to trace CHIP code execution plus constraint posting and waking. Constraint debugging is hard to achieve directly with the debugger and often I have to modify the code and then rerun it several times. A tool might be developed to check code and display information on objects, classes, predicates and graphics.

D. D.: since CLP(FD) is a Prolog-based system, the classical Prolog debugger is the simplest tool to use. However, it does not inform very much on the state of the constraint store neither on the propagation progress. Since the core of CLP(FD) is written in C, a C debugger (gdb / dbx /...) can be used to trace de propagation process.

A. C. and **A. F.**: we use the Prolog box model debugger for pieces of code without constraints. Write statements in the source code are used to:

- locate the problem;
- spy the domains of particular variables.

It is often useful to trace memory consumption as execution proceeds (a dramatic increase is a significant symptom). If the symptoms remain, we try to reduce the problems down to minimal sizes at which they still occur (by reducing the number of variables and removing constraint subsets).

Using application-oriented graphics also proves quite helpful:

- it is easier to see what is wrong with a given solution when it is displayed graphically than in textual form;
- if the graphics are augmented with a propagation-driven animation, the dynamic picture is very useful, in particular for performance debugging;
- it allows indeed to visualise the program making some decisions obviously wrong, thus it gives hints about possible mistakes or propagation weakness.

T. C. : for the constraint part I use tracing via standard output, for the labelling part I prefer graphics.

P. G.:

- a) is used in order to locate the position of any serious error causing the abortion of the execution : it is often the fastest solution to determine the wrong predicate in a big program. It is also used to trace a large number of variables values during an execution, because the "write" predicate is generally fast enough. I often use conditional write, in the following form :

  (condition -> write(something; true)

  This allows to show very specific parts of the execution of a program.

- b) is used to follow a small part of the execution of the program, where you already know there is a problem and where you want to check out the predicates being called and the value of parameters. It can also help to understand the logical behaviour of a program and the reasons for failures.
- if you mean by "formal methods" checking again and again the source code, I should say that it is the most used method when debugging. In fact, it is always used, and that's the reason why it would be very nice to get a tool showing the source code while running a program, such as any C/C++ debugger.

C. L. P.: I put a lot of assertions in the program. Sometimes I run well-known test cases under different scenarios, i.e. I check that the same constraints, given in a different order bring the same propagation results, and the same optimisation results when full optimisation is feasible (small test cases).

K. J.: I add a lot of redundant pre/post checks to my code (e.g. using assert in C++). My most common debugging involve run the coding, receiving an assertion exception. I occasionally use dbx or dbxtool to find where or why the assertion exception occurred. I prefer using tracing on stdout over dbx or dbxtool because with most debugging tools it is to onerous to selectively get more or less information. I.e. you have to manually set/remove breaks, watch points, etc. I use trace statements with run-time flags so that after adding a trace statement, I can selectively enable or disable it a run-time. I find this to be a more effective means of getting more or less detail.

2.6 *Type of bugs and their frequency: Q6*

We come to one of the most relevant questions: which bugs affect constraint programs? We proposed the following set of alternatives:

a. Typos

b. Unexpected term structure or type

c. Wrong constraint declaration

1. Missing constraints
2. Too many constraints

d. The intended semantics of the program are not respected

e. Data inconsistency

f. Other

We asked to rate the frequency of each one according to the usual scale:

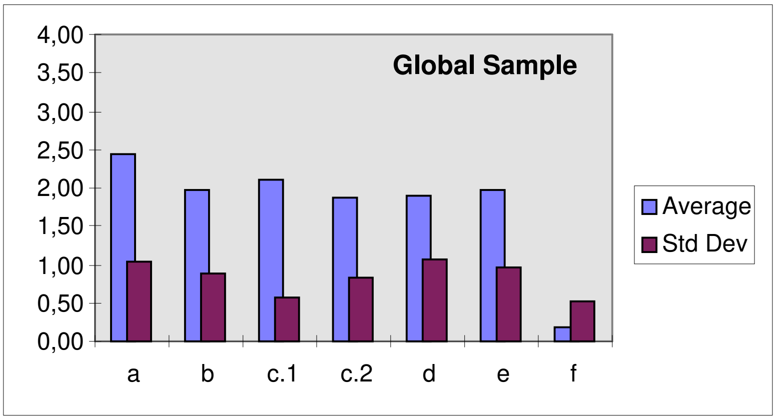
0: Never

1: Seldom

2: Sometimes

3: Frequently

4: Very frequently



The most frequent type of error is the most stupid and simple to correct: a typo in the name of the predicate or in the name of the variable can produce unpredictable behaviours. All the other errors are considered significant: errors of type b) and d) are usually due to a wrong implementation of a procedure or predicate, although b) is sometimes a direct consequence of a typo itself. Errors of type c) has to do with a weak or over-constrained formulation of the model. A further distinction is necessary in the case of data inconsistency. This type of problem is much more significant in the case of operational applications developed in an industrial environment than in the academic environment: the average rating in the former is 3.4 vs. the average rating of 2 in the latter. This is obviously a consequence of the data-sets used in the two environments. For example, benchmarks used at the university include coherent data sets, while industrial applications face the problems of changing data sets.

Comments

P. G.: most errors are due to :

- misspelled name for a variable (typographical error), always stupid but sometimes hard to find out;
- wrong type for a parameter, especially in huge global constraints where parameters can be quite complex lists;
- recursion difficulties, which make a program behave in an unexpected way (Prolog bugs);
- modelling problems with constraints: forgetting some constraints or posting to many ones, or difficulty for mapping complex constraints to physical problems (quite frequent when using "diffn" constraints with more than 3 dimensions...);
- difficulty to understand the way our own heuristic behaves and to guide the search for a (good) solution (i.e. search tree control problems);

- memory management (stack overflow).

C. L. P.: errors in user-defined constraint propagation demon, errors in the code implementing the search strategy (in both cases, the intended semantics of a sub-program are not respected).

2.7 Difficulties encountered: Q7

Which aspects of CP applications development are more difficult to face? This point is fundamental because these aspects probably need better debugging and monitoring tools to tackle them. The proposed answers are a sort of recipe for constraint programming: to develop a CP application you need to understand the basic constraint propagation in your constraint network and how this reduces the domain variables. This understood, you have to find out the best labelling strategy that searches the solutions in the fastest way. This achievement usually requires a tuning and optimising phase. Hence the proposed aspects were:

a. understand constraints propagation

b. understand domains reduction

c. understand the search space traversal

d. tune and optimise the program

e. Other

This time the rating is about the fixing difficulty:

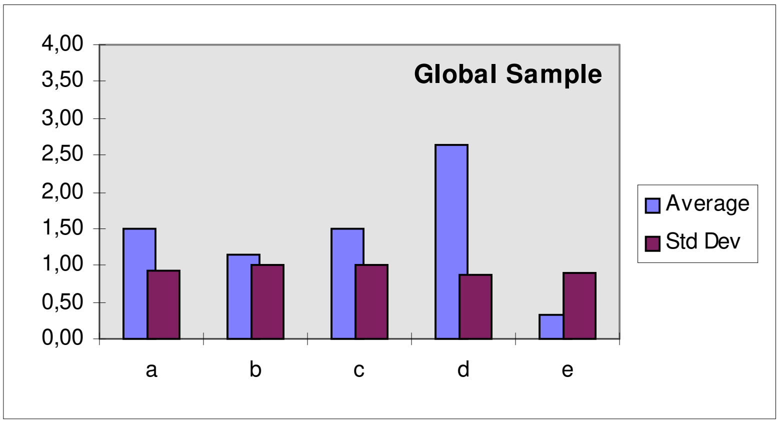
1: Not a problem

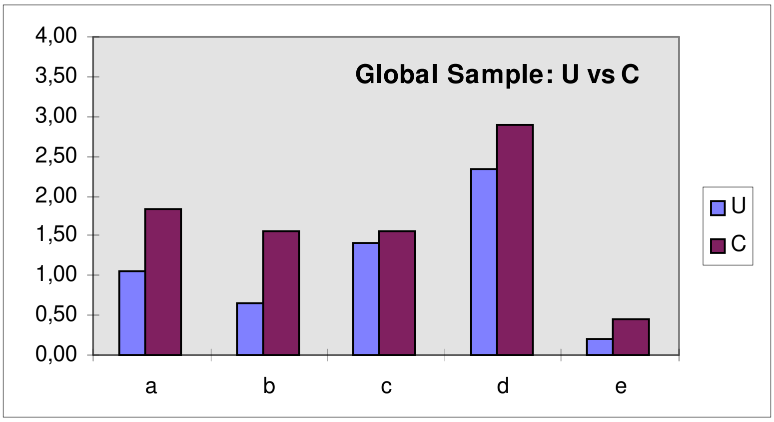
2: Not a big problem

3: Quite a problem

4: Very annoying problem

5: Blocking problem





The most crucial aspect of constraint programming is to improve the performance of the program. This is not surprising since the attacked problems are inherently hard. Writing efficient programs require training and experience because in CP "there is not strict separation between modelling and implementing tasks" ([SK95]) . Writing in a very short time a simple program which is correct but inefficient is a common experience for a CP programmer. This is a "bug" in a broader sense: the program does not work as expected or desired: the modelisation is correct but the program does not find the solutions in the required time limits. We call the process of solving this problem "Performance debugging".

Also the other aspects are recognised as relevant. We note a certain gap between the average ratings of the industrial and the academic sample. This gap might suggest that the basic steps of constraint propagation and domain reduction require a certain knowledge.

Comments

D. D.: "the main problem is not to understand the propagation but really to trace it and detect where/when/why it fails".

A. C. and **A. F.**: debugging the program with respect to its intended semantics is one issue, but perhaps not the hardest. One may not find immediately a correct set of constraints representing the problem specifications. Errors may occur due to a wrong usage of some arguments in global constraints... More difficult is the problem of user-data inconsistency, i.e. of data not satisfying the specified hard constraints. This will often require some pre-processing of the data and/or an alteration of the problem specification. Even harder is performance debugging, when solutions are very slow to obtain for real-sized data, or the system runs forever without providing solutions, or runs out of memory. The most likely cause is a constraint model leading to weak propagation, and/or poor heuristics.

T. C.: the most annoying errors are due to changing data sets, when the program is in use daily.

E. V.: maintenance is a relevant problem not mentioned in the questionnaire.

2.8 *Your ideal C(L)P debugging environment: Q8*

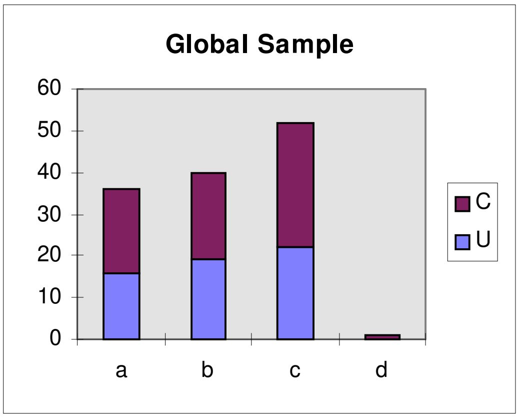
We asked to indicate the preferred debugging environment. Because of our incomplete description, some people indicated more than one answer and gave a rating of preference (e.g. 1, 2, 3). We summarised these heterogeneous indications assigning one unity to each sign of preference. We are confident that the resulting graph captures the intentions of the sample.

a. Static debugging (at compile-time)

b. Text-oriented interactive debugging

c. Graphic-oriented interactive debugging

d. Other



There is a strong demand for graphical tools, which are considered the best way to have an intuitive, synthetic and dynamic description of what is happening. However, graphics are not the panacea of all CP debugging problems: what is also needed is an improvement of the traditional textual debuggers, enhancing the interaction with them and the information they can display.

Comments

M. C.: I think that we need graphics to organise and display the information from the debugger in an easily understandable way. On the other hand, we also need a tool that allows to:

- set dynamically constraints: most of the time it is difficult to test the effect of a new constraint without restarting completely the solver (declare domain variables, posting constraints, etc.). Given a set of variables and constraints on them, it should be possible to enter incrementally a new constraint via a command line and to see the effect on domains.

- show simply why a constraint failed : constraint debugging (unlike constraint posting) is still a matter of specialist knowing what are the propagation and domain reduction mechanisms. Explanation mechanisms could help greatly the "normal" developer.

D. D.: the best environment would be a graphical, with tools to follow the propagation step. It should provide a way:

- to inform the user when a fail occurs on a certain variable/constraint;

- to visualise domain reductions (intermediate values);

- to tune the code to improve its efficiency (may be through a profile information: where are the bottlenecks, where the propagation is not efficient, etc.) .

A. C. and **A. F.**: the finite domains global constraints are the real issue. No industrial application in the area of resource management or logistics can be completed without them. They are very expressive and efficient but, since they encapsulate sets of variables and complex propagation, their utilisation is very hard to debug. Hence we need graphic tools providing some insight into the behaviour of the global constraints and making it possible to see when deductions carried out by the constraint allow to fix parts of the solution. This is dependent on the global constraint.

In general, tools are required for each constraint in order to

- visualise the behaviour of the constraint in terms of deductions carried out internally, e.g. show obligatory areas with "cumulative", or allow the constraint to be violated to some extent and visualise the violation; such tools could be given also in form of example programs to be connected to application-oriented graphics;

- provide profiling information about particular constraint instances and variable instances. This will help performance debugging by showing where the program wastes its time, which blind decisions it makes, etc., thus giving ideas on which propagation to reinforce (if possible) by adding redundant constraints, using additional arguments in the global constraints, etc. ;

- how to design relevant heuristics based on the bottlenecks. This will also ease the identification of global inconsistencies.

It is not clear to us what could be done about the linear solver. Visualising the evolution of n-dimension polytopes as new constraints are added seems hardly feasible.

CLIP group (UPM): in our opinion, an ideal debugger would be a combination of formal techniques with graphical ones. The formal techniques could include things like type checking, inference of properties and, more generally, assertion checking and generation. Basically, it would involve adding an assertion language capable of expressing both the types of assertions given by the user and those generated by the compiler. Checking of assertions could be done in part at compile-time and in part at run-time.

Graphical techniques should present the search space of predicates, domains of variables, and the relationships among constraints and variables are depicted in an intuitive way (this of course does not rule out having a text-based debugger as a backup).

A viewer of the search space would show mainly the control paths of the program, i.e., the program execution “shape”. This can be very useful in debugging programs for efficiency. Thus, it would be useful in order to have an idea of the amount of work underneath each control path.

Showing variable values, and current variable domains (and how execution affects them) would also be great. The interface should allow selecting the variables to show, and the domains could be depicted perhaps using two-dimensional graphs or bar charts, or any other means that clearly depicts them in an intuitive way. These would be dynamic, in the sense that they would change as execution proceeds (this could be driven by the user --e.g., execution would proceed only if the user "clicks some button").

The relationships and/or constraints among variables is another (and probably the most) important thing. As in the previous case it would be required that it be possible to select the variables to be shown in some way. Some constraints can also be abstracted away. For example, instead of showing the actual constraints a generic name could be given to a set of them in order to abstract them (it would be a good idea to use for this the name of the predicate which actually placed those constraints). This would be one level of abstraction, but different abstraction levels on the constraints and variables shown could be defined (or, better, the user could be allowed to define the abstraction).

Another nice feature would be to be able to show the actual code under request. And also, why not, to have an easy-to-use interface ...

P. G.: the main features of a debugging environment should be

- a graphical interface allowing visualisation of the constraints, domain variables and of the search tree;
- interactive tracing and control of the execution of a program: possibility for the user to skip some predicates calls, to run a goal during the execution of a program, etc.

X. S.: my ideal CLP debugging environment is based on the Prolog box model: ENTER, FAIL, RETRY, CUT. It should have “tool functionality” like :

- the positioning of cursor at the current line of source code;
- if error, NEVER aborting: positioning at the place of error;
- possibility to trace or stop on:

  1) methods, procedures or predicates;

  2) particular lines of code;
- possibility to load/relink dynamically correction code.
- control of point of execution:

  <UP/DOWN> functionality to move along execution stack;

  <RETURN> functionality: ability to set manually the return value of current execution stack point (i.e. : procedure or method or predicate). In that case, the execution resumes;

  <FORCE (or RESUME)> functionality: ability to resume execution at current stack point;

  <SET> functionality: ability to modify any global variable or local variable, even on execution on stack;

  <CALL> functionality: ability to call any piece of code from stop point (Rq: debugger can of course handle several imbedded levels of call).

In the Prolog case, several peculiarities:

- setting of variables should include unification (implemented on a local stack?);
- UP/DOWN can be performed:
  1. on a choice point stack;
  2. on the predicate call stack;
- UP/DOWN performs backtracking/redoing => a trailing problem.

On CHIP specific issues:

- choosing which Fails are interesting to trace/stop remains a key problem, also in a C++ CP environment. I wonder which tools we can provide to help the users;
- in case where a FAIL is identified, display of the last assignment invocation (indomain for Prolog), display of last failing constraints:

  a) X #= Value, with value out of domain => display of "X{domain of X before invocation} #= Value";

  b) propagation of elementary constraint failed:

  X #= Value possible but not coherent with X #> Y post before => display of "X{domain of X before propagation} #> Y{domain of Y}"

  c) propagation of a GLOBAL constrained failed. Complete reason can not be given but:

  => display of global constraint invocation, with all current domains of involved variables printed.

T. K.: it would be best to catch all errors at compile time (i.e. with static debugging techniques). However, we all know that this is impossible.

J. M. S.: I've thought of this for only 10 minutes, but if you could graphically display a network of constrained variables, connected by arcs, and have current variable domain information available when someone selects a variable, that would very interesting to me to work with. This chart would be available and updated at program breakpoints.

2.9 Comments and examples: Q9

CLIP group: analysis of appropriateness of flow control primitives. Common advice on control usage can not be directly lifted from Prolog to CLP:

max(X, Y, Y):- X < Y, !.

max(X, Y, X):- X >= Y.

This code, in absence of runtime errors, has the same behaviour with or without the "cut" in Prolog. In a CLP language, the first clause might merely add a constraint, and thus its completeness cannot be ensured without prior data-flow analysis of the program. This is a possible example of "missing solutions".

B. D.: a great improvement in debugging tools would be to keep the trace of a program even if a demon callback is called, and to be able to:

- follow the source code;
- put and withdraw graphically stop points;
- see and modify domain variables;
- see and modify variables;
- see, modify, add and forget constraints;
- interrupt a goal, try another one while keeping the results of the first, than get on with the first goal.

C. L. P.: average (and largest) number of variables and constraints (question 3.2.1) are not really meaningful. For example, how do you count a set variable? As only one set variable or as n 0-1 variables?

Concerning question 4. Compile-time errors are the most frequent, but they generally are very easy to repair. Missing answers are much more difficult to fix. "Long to find answers" often occurs because the search strategy is bad, but less frequently because there is a bug (in this case, it often goes with missing answers).

Concerning question 7: d is really about efficiently solving hard problems, not much about debugging. a/b/c are OK if the underlying tool and its documentation are sufficiently precise. However, when the user develops his or her own constraints, the corresponding propagation procedures may contain bugs, which tend to be the most difficult to detect.

Concerning question 8. The more you can do at compile-time, the better. But it seems it would be very difficult to develop formal compile-time techniques to detect that the intended semantics of a sub-program are not respected.

M. R.: debugging must also be possible in runtime environments when linked to c++ code.

3. Debugging Needs

We think we can collapse the multiple emerging needs of debugging in two main categories:

- *correctness debugging*;
- *performance debugging*.

The rest of this section is devoted to illustrate the characteristics of these two categories.

3.1 Correctness debugging

By *correctness debugging* we mean the actions the programmer does in order to make the program behave as expected, i.e. according to his/her intention. The ultimate goal of this debugging activity is to locate all the errors and faults which are revealed by **missing/incorrect answers** (*semantics errors*). Note that the missing answers problem resolves itself into two formidable (and undecidable) problems: detecting the reason of failure and detecting non termination.

As we have seen, the potential sources of missing and incorrect answers are: typos in the program text, wrong typing, wrong design and coding of modules and procedures, wrong constraint modelisation. Finding these errors is the traditional debugging activity, but in the CP framework this is somehow complicated by the fact that the execution of the program is data-driven. Hence traditional step by step, source oriented and trace oriented approaches are insufficient.

Sometimes, however, the problem is not in the program itself but in the consistency of the data set on which it must operate. If we include in the semantics of the program all the possible data sets on which it is intended to operate, we can see how **data-inconsistency** can be seen as a special case of correctness debugging: the semantics of the data are inconsistent or incompatible with the semantics of the program. It is worth noting that this time it is not the program which must be debugged but the data-set. However the symptom is likely to be the same: a missing or incorrect answer, and in general an unexpected behaviour of the program. Whatever the tools we are using for debugging the semantics of the program, they should end up to be helpful to locate that the problem is on the data (although fixing it or guarantee data consistency is another issue).

3.2 Performance debugging

Performance debugging has been posed as a very relevant problem. Constraint programs must be not only correct but also efficient. This is very often a primary requisite for a constraint program. As stated in [M95a]:

> ''Developing CLP applications is a difficult task. This is to a large extent due to the fact that CLP is usually being applied to combinatorial search problems for which no efficient algorithms are known. Instead of following some well-known rules, the users have to experiment with various models, approaches and strategies to solve the problem. Apart from handling the usual *correctness debugging* of their programs, users are also facing the problem of *performance debugging*, i.e. finding a way to execute CLP programs efficiently. To date, there exists no satisfactory methodology for de-

bugging CLP programs. There are basically two ways to approach the problem: either try to apply all available methods exhaustively, last resort being the simplification and downscaling of the problem , or to analyse the behaviour of the program and try to understand what is the reason for its poor performance. The current CLP systems unfortunately offer little support for this task and there are not widely available tools which would support tracing and (performance) debugging CLP programs."

Most of the times, bad performances are due to a poor comprehension and modelisation of the problem. Unfortunately this aspect is sometimes obscured by the constraint solver. As a first consequence of the problem of performance debugging, there is a need to understand constraint propagation. Points to investigate are if and why

- constraints are not waked when expected;
- constraints are uselessly waked (small or no reduction of domains).

However the problem it is not only to understand how propagation works, but also how propagation fails. Finding the reason of failure is useful in trying to obtain efficient pruning of the failing search alternatives.

Performance debugging rises also the need to understand the search space traversal. Again, finding very quickly a first set of solution and then running a long time for new ones is a very common behaviour for a CP optimisation program execution. The programmer is left with the doubt if the program is lost in a local search or if the search space does not contain any better solution. Even when he/she realises that the first case applies, very few information is provided on how he/she could change the labelling strategy.

4. Addressing the needs

In this section we describe why and how (in our opinion) the three main streams of research and development of the project (i.e. graphical tools, assertion based tools and declarative diagnosis tools) address the emerging needs and how these tools can co-operate.

4.1 Graphical tools

Graphics can be very intuitive and can convey a very abstract and concise description of what is happening in the computation. They are very well suited for the description of the dynamic behaviour of the program. For all these reasons they can provide significant help both on performance and on correctness debugging. Moreover, as we will see, they can be used to support more advanced tools such as assertion based or declarative diagnosis tools. To satisfy all these multiples objectives, they must:

- describe the search tree, the labelling strategy and how the control affects them;
- show the values of variables to visualise and understand domain reduction and constraint failure;
- show relationships among variables to describe the constraint semantics, interaction and propagation;
- allow multiple level of abstraction to gain a very precise view or a global picture, according to the need of the moment and the subject of the visualisation.

4.2 Assertion based tools

Assertion based tools tackle the problem of correctness debugging according to the following schema: given an assertion language (to be defined)

1. the programmer declares his/her intentions about the behaviour of the program through a set of assertions;
2. the system verify if they are valid or if some computation could violate them;
3. another set of assertions is generated automatically by the system;
4. the system and the programmer relate the two kinds of assertions and check their consistency;
5. if one inconsistency is detected, the programmer's intention and the program behaviour do not correspond to each other.

4.3 Declarative diagnosis tools

The other approach to tackle the problem of finding *semantics errors* relays on the property of many CP programming languages to be declarative: their semantics are independent of their execution model. This implies that also the notion of error is declarative: a small piece of code can be identified as erroneous without referring to the operational semantics. The basic idea of declarative debugging is that to debug an incorrect program, all a programmer needs to know is *his/her* intended interpretation of the program. In particular, knowledge of the procedural behaviour of the system running the program is unnecessary. This is a very attractive property since:

> "constraint programming is inherently different from imperative, functional or Prolog-type logic programming, since it is non-algorithmic. Even in Prolog (without delays), and more so in conventional languages, the program execution follows a fixed scheme which implements a particular algorithm, no matter if the program is declarative or not. CP however, only states constraints and then looks for a solution that satisfies them. The search is rather data-driven than program-driven " ([M95a]).

Quoting again [M95a]:

> "Similarly to the search for a solution which is performed by the program, the debugging itself is also a search problem. The debugger to support it must therefore be highly interactive and open".

The search structure and the high degree of interaction are the core properties of Declarative Debugging: the programmer gives a symptom of a bug to the debugger, which then interactively poses to the programmer a set of questions about the validity of certain expressions in the intended interpretation. If the process is successful, the debugger finally presents an incorrect statement to the programmer.

The main problem of this approach is that the programmer should be able to state the validity of expressions which can involve a huge number of constraints. Hence the basic need to address in this case is the possibility to represent complex computational states in a understandable way. It is worth noting that graphical tools and assertion can be of great help in this very ambitious task.

5. Conclusions

Several positive conclusions can be drawn from this summary of the questionnaire data. Almost all questions have provided significant and coherent answers. These answers globally capture the multiple aspects of CP development. The consortium includes all different "souls" of the current CP community: academic researchers, industrial platforms developers and sellers, professional end users. All these different actors exchange information and experience in a very short time.

As it is clear from the result of question Q8 and from the collected comments, users express a strong demand for graphical tools. It is not a novelty that they can be very helpful. All current CP platforms provide (or can be linked to) graphical environments. Hence users already experienced the usefulness of a graphical description. The main goal of the project here is hence to provide tools which are flexible, powerful and conceptually new, instead of leaving the users to program his/her own.

A different perspective must be taken in the case of assertions and declarative debugging. Here the project aims to be more experimental and research oriented. In the case of assertions, the point to understand is if some research topics (assertion verification and synthesis, abstract interpretation) have reached the level of maturity to provide effective tools in the difficult context of industrial constraint programming. In the case of declarative debugging the approach is even more oriented toward pure research: this attractive schema must attack the problem of abstracting complex computational states to let the programmer decide about their correctness.

6. References

[CFGG95] A. Chamard, A. Fischler, D. B. Guinaudenau, A. Guillaud. *CHIC Lessons on CLP Methodology*. Deliverable D2.1.2.3. ESPRIT Project EP5291 "CHIC: Constraint Handling in Industry and Commerce", 1995

[M95a] M. Meyer. *Debugging Constraint Programs*. Technical Report ECRC-95-15, ECRC, 1995

[M95b] M. Meier. *Debugging Constraint Programs*. In Principles and Practice of Constraint Programming-CP'95, Cassis, LNCS 976, Springer Verlag, pp 204-221, Sept 1995.

[SK95] H. Simonis, P. Kay. *Application Development with the CHIP System*. Commercial Report, COSYTEC SA, October 1995

End of the document